Universidade do Minho
Escola de Engenharia

Departamento de Informática.

Licenciatura em Engenharia Informática

23 de Março de 2015

Programação em Lógica

TRABALHO PRÁTICO – 1º EXERCÍCIO

SISTEMAS DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO

Orlando José Gomes Martins Costa a67705

Paulo Ricardo Cunha Correia Araújo a58925

Rui Pedro Azevedo Oliveira a67661

I. Resumo

O presente documento apresenta um sistema de representação de conhecimento e raciocínio, cujo propósito é caracterizar um universo de discurso com o qual se pretende descrever uma árvore genealógica de uma família, e serve de suporte ao trabalho desenvolvido.

Este documento encontra-se estruturado sob a forma de um relatório de desenvolvimento técnico, e reflete o trabalho realizado ao longo da 1ºfase da componente prática da unidade curricular de Sistemas de Representação de Conhecimento e Raciocínio.

Inicialmente é efetuada a descrição das estratégias definidas para representar a informação pretendida. De seguida são apresentadas as tarefas realizadas, assim como as decisões que orientaram a sua implementação.

Finalmente são apresentados exemplos de utilização da base de conhecimento assim como os seus resultados, e é realizada uma análise em forma de conclusão acerca do trabalho desenvolvido e dos seus pontos-chave.

Conteúdo

II. Introdução

O propósito deste trabalho reside na fomentação e melhoria das competências na programação em lógica e com a linguagem PROLOG.

Para isso foi desenvolvido um sistema de representação de conhecimento e raciocínio baseado nas funcionalidades necessárias à solução das 5 questões propostas, apresentadas seguidamente.

O sistema desenvolvido possui então a capacidade de caracterizar um universo de discurso com o qual se pretende descrever uma árvore genealógica de uma família. Isto é alcançado através da inferência de predicados lógicos e invariantes numa base de conhecimento, correspondentes às diversas relações familiares (dependentes e independentes), assim como a atributos dos elementos em questão, tais como a naturalidade. O objetivo final do sistema é então possuir a capacidade de representar conhecimento assim como resolver problemas através da construção de mecanismos de raciocínio.

1 Pergunta 1 – Naturalidade

1.1 Análise do problema

De forma a entender o que realmente acontece no mundo, precisamos entender o que não acontece, isto é, as propriedades invariantes das entidades ou objetos que o compõem. Com essa ideia em mente, podemos considerar lógicos os conjuntos de declarações que possuem a propriedade de ser verdadeiros ou falsos independentemente do tempo ou lugar que ocupam no universo considerado. Sendo assim, de forma a impedir certas situações que não se verificam no mundo real, desenvolvemos uma série de invariantes.

Com o objetivo de impedir a representação de situações que não acontecem na realidade, foi necessária a criação de invariantes relacionados com a naturalidade de um indivíduo. Os invariantes desenvolvidos garantem que um indivíduo não tem mais do que uma naturalidade e que o seu ano de nascimento nunca é superior ao ano da sua morte.

1.2 Implementação

Primeiramente, começamos por encontrar e colocar todas as naturalidades L, associadas a um indivíduo A numa lista S, através do uso do predicado soluções (findall). De seguida, exigimos que essa lista S tenha comprimento menor ou igual a 1.

+**naturalidade(A,L,AN,AM)** :: (solucoes(X,naturalidade(A,X,AN,AM),S),

comprimento(S,N),

N =< 1

).

(I.1)

Através do seguinte invariante, garantimos que o ano de nascimento (AN) de um indivíduo nunca é superior ao ano da sua morte (AM).

+naturalidade(A,L,AN,AM) :: AN =< AM.

(I.2)

1.3 Exemplificação

1º Exemplo - o ano de nascimento é maior que o ano de morte - não é possível inserir

```
| ?- evolucao(naturalidade(a1,'guimaraes',1010,1002)).
```

no

2º Exemplo: toda a informação esta correta - é possível inserir

```
| ?- evolucao(naturalidade(a1,'guimaraes',1010,1020)).
```

```
 yes
```

3º Exemplo - a informação já se encontra na base de conhecimento, não e possível inserir

```
| ?- evolucao(naturalidade(a1,'guimaraes',1010,1020)).
```

```
 no
```

2 Pergunta 2 – Relações Familiares Independentes

2.1 Análise

Primeiramente, começamos por impedir situações em que é inserido conhecimento repetido, ou seja, caso exista na base de conhecimento uma relação filho (A,B), esta relação não pode voltar a ser inserida.

De seguida, quisemos garantir que situações como um indivíduo A ter 5 avós sejam impossíveis. Então, foram criados invariantes de forma a garantir que um indivíduo apenas pode ter 2 pais e 4 avós.

Outro tipo de situações que não pode existir é, por exemplo, que um indivíduo A, filho de B, possa ser pai/avô do mesmo indivíduo B. Esta é uma situação impossível de acontecer no mundo real, logo não deve ser possível a sua representação.

2.2 Implementação

Com este invariante pretendemos garantir que não é possível a inserção de conhecimento repetido. Em primeiro lugar, procuramos e colocamos todas relações filho-pai entre os indivíduos F e P numa lista S. De seguida, garantimos que existe apenas uma ocorrência. O mesmo se aplica ao invariante +neto, representado abaixo.

```
+filho( F,P ) :: ( solucoes( (F,P), (filho( F,P ), S ),
                  comprimento( S,N ),
                  N == 1 ).
```

(I.3)

```
+neto( Ne,A ) :: ( solucoes( (Ne,A), neto( Ne,A ), S ),
                    comprimento( S,N ),
                    N == 1 ).
```

(I.4)

Procuramos e colocamos numa lista S1 todos os indivíduos X que sejam pai de F. De seguida, através do predicado “únicos”, removemos todos os elementos repetidos da lista anteriormente obtida. Por fim, garantimos que o comprimento dessa lista é inferior a 3, ou seja, o indivíduo F apenas pode ter 2 pais.

O invariante “+neto (Ne,A)” tem o mesmo propósito, apenas com a diferença de o indivíduo Ne não ter mais do que 4 avós.

+filho(F,P) :: (solucoes(X, (filho(F, X)),S1),

unicos(S1,S),

comprimento(S,N),

N < 3

).

(I.5)

+neto(Ne,A) :: (solucoes(X,neto(Ne,X),S),

comprimento(S,N),

N =< 4

).

(I.6)

Através deste invariante garantimos que quando pretendemos inserir uma nova relação familiar entre dois indivíduos A e N, estes não têm qualquer relação entre eles. Por exemplo, quando pretendermos alargar a nossa base de conhecimento dizendo que um indivíduo A é avô de outro indivíduo N, não deve ser possível que N já tenha outra relação familiar com A. O mesmo se aplica à relação pai-filho.

+avo(A,N) :: nao(pai(A,N)).

+avo(A,N) :: nao(filho(A,N)).

+avo(A,N) :: nao(neto(A,N)).

+neto(N,A) :: nao(filho(N,A)).

+neto(N,A) :: nao(pai(N,A)).

+neto(N,A) :: nao(avo(N,A)).

+pai(P,F) :: nao(avo(P,F)).

+pai(P,F) :: nao(filho(P,F)).

+pai(P,F) :: nao(neto(P,F)).

(I.7:I.15)

2.3 Exemplificação

1º Exemplo - a informação esta correto, insere

```
| ?- evolucao(pai(a,c)).
yes
```

2º Exemplo - a informação já se encontra na base de conhecimento (pai(a,c)), não insere

```
| ?- evolucao(filho(c,a)).
```

no

3º Exemplo - a informação esta correta

```
| ?- evolucao(filho(c,b)).
```

yes

4º Exemplo – ‘c’ já tem 2 pais na base de conhecimento, não insere.

```
| ?- evolucao(filho(c,d)).
```

no

5º Exemplo – ‘a’ é pai de c logo não podem ser irmãos, não insere

```
| ?- evolucao(irmao(a,c)).
```

no

3 Pergunta 3 – Primos, Tios e Irmãos

3.1 Análise

Esta questão levanta mais problemas, dado que para maximizar as respostas o sistema terá que ser capaz de calcular as relações familiares, não apenas através do seu próprio predicado, mas também através de outros predicados.

O cálculo destes predicados derivados deve ser tratado com algum cuidado devido a que o PROLOG facilmente poderá entrar em ciclo infinito.

Assim, escreveram-se todas as fórmulas lógicas que relacionavam os predicados. E depois corrigiu-se os ciclos infinitos analisando as dependências entre os predicados. Desta análise das dependências surgiu o seguinte diagrama:

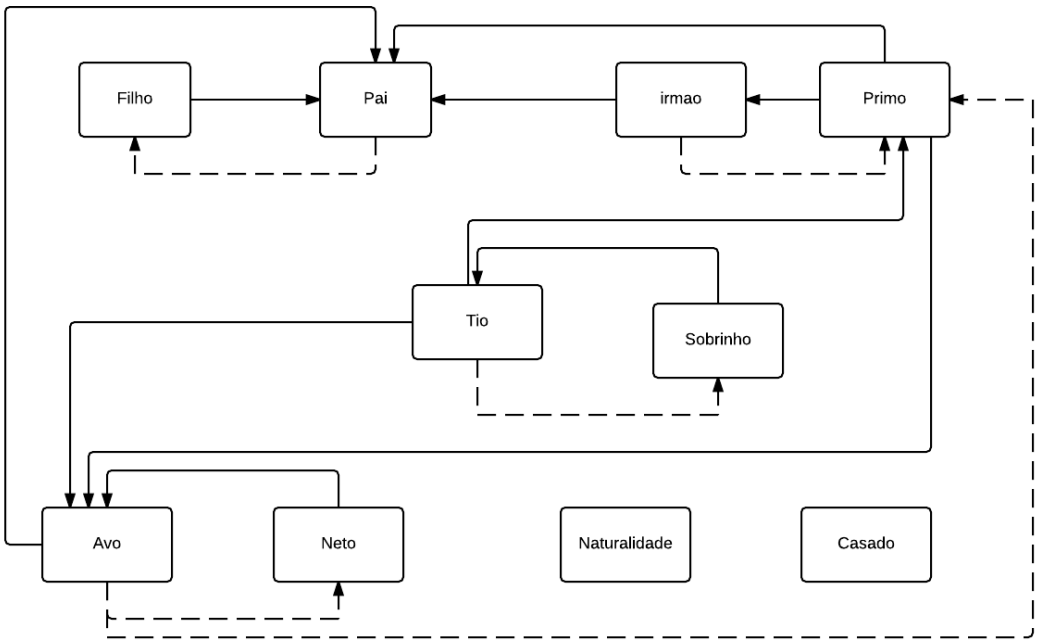

Figura 1: Diagrama de dependências

Setas continuas: Representam dependências de todo o predicado.

Setas a tracejado: Representam dependências apenas dos factos do predicado.

Inicialmente neste diagrama todas as setas estavam a negrito (dependências totais). De seguida alterara-se para dependências dos factos apenas, até que as dependências totais deixassem de formar qualquer tipo de ciclo.

3.2 Implementação

De forma a consultar os factos foi utilizado o predicado “clause”, que recebe como argumentos a cabeça e o corpo do predicado que no caso dos factos será *true*.

Definiram-se os predicados que são inversos uns dos outros, sendo que este género de predicados está implementado de forma a não entrar em ciclos e não gerar informação repetida, pois o predicado do filho apenas é verdade quando este não é já um facto.

pai(X,Y) :- clause(filho(Y,X),true).

(P.1)

filho(X,Y) :- nao(clause(filho(X,Y),true)), pai(Y,X)

(P.2)

…

De seguida exprimiram-se possíveis derivações de informação para cada um dos predicados, tendo em atenção a análise desta questão de forma a não entrar em ciclos.

Criou-se o predicado auxiliar “tester” responsável por testar cada um dos predicados auxiliares. Este predicado foi criado para tentar evitar alguma informação repetida. Desta forma apenas são executados os predicados até um deles ser verdade (devido ao problema de várias cláusulas poderem ser verdade ao mesmo tempo). No entanto existe ainda o problema não solucionado, sendo este o caso de “pai(X,A), primo(Y,A)” onde existe mais do que um A capaz de tornar esta clausula verdadeira.

tio(X,Y) :- nao(clause(tio(X,Y),true)), tester([tio1(X,Y),tio2(X,Y),tio3(X,Y)]).

(P.3)

tio1(X,Y) :- irmao(X,A), pai(A,Y), X\=Y.

(P.4)

tio2(X,Y) :- pai(X,A), primo(Y,A), X\=Y.

(P.5)

tio3(X,Y) :- avo(A,Y), pai(A,X), nao(pai(X,Y)), X\=Y. %por a ordem inversa não funciona

(P.6)

O predicado “tester” consiste na lógica de que caso o predicado da cabeça da lista seja verdade devolve *ok*, caso contrário continua a testar.

tester([H|T]):- H.

(P.7)

tester([H|T]):- tester(T).

(P.8)

A definição de todos os predicados pode ser encontrada em anexo.

Com todas estas relações para a implementação dos predicados primos, tios, irmãos entre outros torna-se muito simples:

primos(X,S) :- solucoes(A,primo(X,A),S1), unicos(S1,S).

(P.9)

tios(X,S) :- solucoes(A,tio(A,X),S1), unicos(S1,S).

(P.10)

...

A utilização do predicado "únicos" (predicado que dada uma lista retorna os elementos que são unicos) deve-se ao problema descrito anteriormente da redundância gerada em certas situações que existe mais do que uma pessoa a tornar o predicado verdadeiro.

3.3 Exemplificação

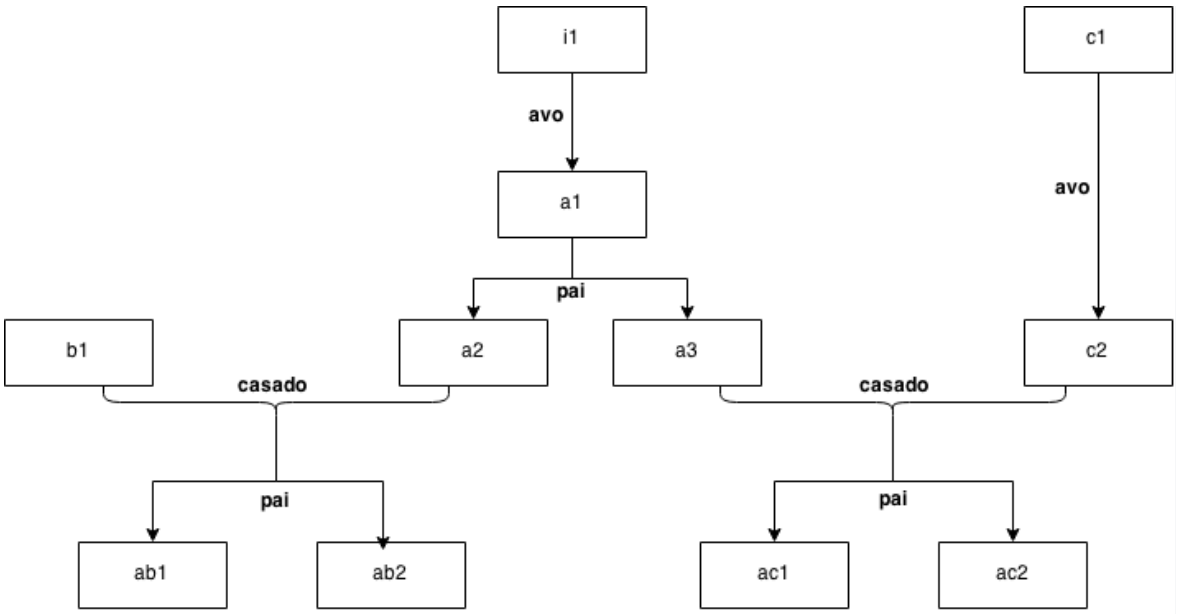

Figura 2: Cenário de estudo

1º Exemplo - cálculo dos tios de ac1

```
| ?- tios(ab1,X).

X = [b1,a2] ?

yes
```

2º Exemplo - cálculo dos primos de ac1

```
?- primos(ac1,X).

X = [ab2,ab1] ?

yes
```

3º Exemplo - calculo dos irmãos de ac1

```
| ?- irmaos(ac1,X).

X = [ac2] ?

yes
```

4 Pergunta 4 – Descendentes e Ascendentes

4.1 Análise

Para o cálculo dos descendentes de um individuo, tornar-se-ão úteis os predicados desenvolvidos na alínea anterior que calculam todos os filhos e netos de um individuo.

Utilizando estes predicados será suficiente criar um predicado recursivo que calcule os descendentes diretos e para cada um dos descendentes continuar a calcular os descendentes.

Também deve ser tido em conta o facto de se poder ser descendente não só através da relação pai-filho mas também neto-avo.

4.2 Implementação

Durante a implementação detetou-se que o "descendentes" deveria ser capaz de calcular todos os descendentes a partir de uma lista. Sendo assim, foi criado um predicado auxiliar ((descendetesAux) para isso.

descendentes(A,N,R) :- descendentesAux([A],N,R1), únicos(R1,R).

(P.11)

Assim sendo o "descendentesAux" terá três casos de paragem, dois deles com o resultado de lista vazia, ou seja sem descendentes. O último apenas calcula os filhos, única relação possível com grau igual a 1.

descendentesAux([],_,[]).

descendentesAux(_,N,[]) :- N =< 0.

descendentesAux([A],1,R) :-

filhos(A,R).

(P.12)

Existem dois casos de recursividade possível, um deles para tratar os descendentes diretos de um individuo enquanto que o outro serve para iterar sobre a lista.

Desta forma, quando a lista tem apenas um individuo, calculam-se os filhos e os netos e calculam-se os descendentes destes filhos e destes netos. Finalmente toda a informação é inserida numa lista.

descendentesAux([A],N,R) :-

N > 1,

filhos(A, F1),

N1 is N -1,

netos(A, F2),

N2 is N -2,

descendentesAux(F1,N1,R1),

descendentesAux(F2,N2,R2),

concatena(F1,F2,A1),

concatena(A1,R1,A2),

concatena(A2,R2,R).

(P.14)

Quando a lista ainda tem vários indivíduos, estes irão ser processados recursivamente, no final concatenando-se o resultado dos descendentes de um indivíduo, com os restantes descendentes de todos os indivíduos da lista.

descendentesAux([A|T],N,R) :-

descendentesAux([A],N,R1),

descendentesAux(T,N,R2),

concatena(R1,R2,R).

(P.15)

O predicado “ascendente” foi implementado de forma semelhante ao predicado “descendente”.

4.3 Exemplificação

1º Exemplo - todos os descendentes de a1 com 2 graus

```
| ?- descendentes(a1,2,R).
R = [ab1,ab2,ac1,ac2,a2,a3] ?
yes
```

2º Exemplo - todos os descendentes de a1 com 1 grau

```
| ?- descendentes(a1,1,R).

R = [a2,a3] ?

yes
```

3º Exemplo - todos os ascendentes de ab1 com 1 grau.

```
| ?- ascendentes(ab1,1,R).

R = [b1,a2] ?

yes
```

4 exemplo todos os ascendetes de ab1 com 2 grau.

```
| ?- ascendentes(ab1,2,R).

R = [a0,a1,b1,a2] ?

yes
```

4º exemplo todos os ascendetes de ab1 com 4 grau.

```
| ?- ascendentes(ab1,4,R).

R = [i1,a0,a1,b1,a2] ?

yes
```

5 Pergunta 5 – Cálculo de Relações Familiares

5.1 Análise

Para o cálculo da relação familiar entre dois indivíduos é necessário um predicado capaz de testar cada uma das relações possíveis, e informe quais destas são verdade.

Assumindo a existência de uma lista com todas as relações da base de conhecimento, seria suficiente percorrer essa lista e dizer quais dessas relações são verdadeiras.

Assim, é necessária uma lista com todas as relações existentes no sistema e um predicado que recebe uma lista de predicados e retorne os que são verdade.

5.2 Implementação

Implementou-se um predicado que cria uma lista de relações, e as unifica com os indivíduos pretendidos, facilitando a adição de novos predicados de relações:

relacoesAux(X,Y,[filho(X,Y),pai(X,Y),primo(X,Y),sobrinho(X,Y),avo(X,Y),neto(X,Y),tio(X,Y),irmao(X,Y)]).

(P.16)

Utilizando este predicado, criou-se o predicado relações, que começa por gerar a lista de predicados em Q e depois testa cada um desses predicados, sendo que o resultado será a lista S. Está previsto que um individuo possa ter varias relações familiares com outra pessoa.

relacoes(I1,I2,S) :- relacoesAux(I1,I2,Q), quais(Q,[],S).

(P.17)

O predicado 'quais' recebe a lista de predicados Q, testa cada um dos predicados e devolve o resultado para S. A sua implementação é a seguinte:

quais([H|Q],R,S) :- H, quais(Q,[H|R],S).

quais([H|Q],R,S) :- nao(H), quais(Q,R,S).

quais([],S,S).

(P.18)

5.3 Exemplificação

1º Exemplo - cálculo da relação entre dois indivíduos conhecidos.

```
| ?- relacoes(a2,ab1,R). R = [pai(a2,ab1)]
```

2º Exemplo - cálculo de todas as relações de um individuo conhecido. Este exemplo contem repetidos devido ao problema explicado na questão 3.

```
| ?- solucoes(R,relacoes(a2,Y,R),S).
S = [[filho(a2,a0)],
[filho(a2,a1)],
[pai(a2,ab1)],
[pai(a2,ab2)],
[tio(a2,ac1)],
[tio(a2,ac2)],
[tio(a2,ac1)],
[tio(a2,ac2)],
[tio(a2,ac1)],
[tio(a2,ac2)],
[tio(a2,ac1)],
[tio(a2,ac2)],
[tio(a2,ac1)],
[tio(a2,ac2)],
[irmao(a2,a3)],
[]] ?
```

6 Conclusões e Sugestões

Durante a realização do trabalho, foram desenvolvidos diversos invariantes que reforçam a consistência dos predicados e estabeleceram-se várias relações entre predicados permitindo associá-los de forma a ser possível derivar relações a partir de outras relações. O cálculo de todos os primos, tios e outros familiares é assim capaz de obter mais respostas, sendo também mais dinâmico e racional. Os predicados correspondentes aos "descendentes" e "ascendentes" foram também implementados fazendo uso de predicados da alínea 3 (tais como filhos, avós, pais e netos), facilitando a sua implementação e aumentando a interligação dos diversos elementos da base de conhecimento e raciocínio. O predicado "relações" possui a capacidade de verificar relações de forma dinâmica, sendo que todas as relações são geradas a partir da unificação dos seus argumentos, facilitando a verificação da veracidade destas.

Como possíveis melhorias ao nosso trabalho, é proposta a implementação de testes automáticos com a capacidade de confirmar rapidamente alterações em predicados, através de testes associados.

No decorrer do trabalho realizado, foram também enfrentados problemas, tais como a existência de ciclos infinitos do PROLOG, tendo este sido resolvido utilizando o predicado "clause" existente também no PROLOG, e analisando as dependências dos predicados, evitando assim a criação dos ciclos. A apresentação de resultados repetidos em vários predicados, nomeadamente os predicados "irmão", "primo" e "tio" quando inquiridos com o predicado soluções, constituiu também um problema, obrigando a alteração do código inicial e uma melhoria na solução (como mencionado na secção da pergunta 3). No entanto esta solução não é ainda "irrepreensível", pois existe ainda um caso que gera informação repetida, e irá ter de ser melhorada nas fases de desenvolvimento seguintes.

Em forma de crítica final, apesar dos problemas encontrados, o grupo sente-se confiante na sua habilidade de cumprir os desafios propostos e crê ter atingido o objetivo subjacente na realização do trabalho, tendo criado um sistema de representação de conhecimento e raciocínio simples mas inteligente.